

On the Statistical Analysis of Practical SPARQL Queries

Xingwang Han^{1,2}
xingwanghan@tju.edu.cn

Zhiyong Feng^{1,2}
zyfeng@tju.edu.cn

Xiaowang Zhang^{1,2}
xiaowangzhang@tju.edu.cn

Xin Wang^{1,2}
wangx@tju.edu.cn

Guozheng Rao^{1,2}
rgz@tju.edu.cn

Shuo Jiang³
jiangshuo@tju.edu.cn

¹School of Computer Science and Technology, Tianjin University, Tianjin, China

²Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin, China

³School of Computer Software, Tianjin University, Tianjin, China

ABSTRACT

In this paper, we analyze some basic features of SPARQL queries coming from our practical world in a statistical way. These features include three statistic features such as the occurrence frequency of triple patterns, fragments, well-designed patterns and four semantic features such as monotonicity, non-monotonicity, weak monotonicity (old solutions are still served as parts of new solutions when some new triples are added) and satisfiability. All these features contribute to characterize SPARQL queries in different dimensions. We hope that this statistical analysis would provide some useful observation for researchers and engineers who are interested in what practical SPARQL queries look like, so that they could develop some practical heuristics for processing SPARQL queries and build SPARQL query processing engines and benchmarks. Besides, they can narrow the scope of their problems by avoiding those cases that do possibly not happen in our practical world.

CCS Concepts

•Information systems → Query languages;

Keywords

RDF, SPARQL, Well-designed patterns, Monotonicity, Satisfiability

1. INTRODUCTION

The Resource Description Framework (RDF) [16], firstly recommended by the World Wide Web Consortium (W3C) in 1998 [14], is a directed, labeled graph data format for representing information in the Semantic Web. RDF, as a graph model [21, 8], is often used to represent personal information, social networks, metadata about digital artifacts as well as to provide a means of integration over disparate sources of information. The SPARQL query lan-

guage released by the RDF Data Access Working Group in 2004 becomes the official W3C Recommendation for RDF query language [19] in 2008. It is an important language in graph databases which use graph structures with nodes, edges and properties to represent and store data [22, 23]. SPARQL allows for a query consisting of triple patterns, conjunctions (AND), disjunctions (UNION), optional patterns (OPT) and built-in conditions (constraints) to be filtered (FILTER). The standard query language for RDF data is SPARQL [20]. Current version 1.1 of SPARQL extends SPARQL 1.0 [19] with important features such as aggregation and regular expressions. Other features, such as negation and subqueries, have also been added, but mainly for efficiency reasons. They were already expressible by a more roundabout manner in version 1.0 (this follows from known results to the effect that every relational algebra query is expressible in SPARQL [14, 24]). Hence, it is still relevant to study the fundamental properties of SPARQL 1.0.

In this paper, we present some statistics of seven basic features on practical SPARQL queries coming from our real world. In particular, we analyze a log of SPARQL queries, harvested from Linked SPARQL Query Log Dataset (LSQ) published in 2015: a public, openly accessible dataset of SPARQL queries extracted from endpoint logs where the DBpedia SPARQL Endpoint is included [17]. The dataset contains more than 1.19 million unduplicated queries in total. These features include three statistic features, namely, *the occurrence frequency of triple patterns, fragments, well-designed patterns* and four semantic features, namely, *monotonicity, non-monotonicity, weak monotonicity* (old solutions are still served as parts of new solutions when some new triples are added) and *satisfiability* which is used to determine whether a query is in error or meaningless. Though there are some existing works in statically analyzing SPARQL queries [15, 1, 10, 9, 6, 4, 17], they either study some semantic features in a qualitative way [1, 10, 9, 6, 4] or just analyze some statistic features [15, 17]. As far as we known, it is still open to analyze these semantic features in a quantitative way. However, we have investigated that it is not trivial to do this since these semantic features of a single query might become totally different from semantic features of fragments. For instance, the fragment AO, whose patterns contain only two operators AND and OPT, is non-monotonic while a pattern $(?x, p, ?y)$ OPT $(?x, q, ?z)$ is weakly monotonic. Moreover, we have investigated that many queries can be determined whether they are satisfiable or not, even they belong to fragments whose satisfiability is undecidable. For

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

instance, a pattern $(?x, p, ?y)$ FILTER $(?x \neq ?z) \wedge (?y = c)$ belongs to the fragment SPARQL($=, \neq$) whose satisfiability is undecidable [24].

The main goal of this paper is to present a comprehensive statistical report about these seven features over SPARQL queries in LSQ. To simplify our discussion, we mainly consider these queries in SPARQL 1.0 (over 99.94%). The main contributions of this paper can be summarized as follows:

- We analyze three statistic features: the occurrence of triple patterns, fragments and well-designed patterns. And then we find the followings: a) over 96 % of practical queries contain at most 7 triple patterns while those queries with at least 8 triple patterns are less than 4%. b) Among 32 fragments of SPARQL 1.0, the four fragments: *none*, A, F and AO, cover over 85% of practical queries and the remaining 22 fragments contain less 15% of practical queries. Additionally, the six fragments: FGO, FGU, GOU, FGOU, AFGU and AFGOU, do not occur in our practical world. c) These practical queries with well-designed patterns known to be weakly monotonic is about 77.66%. About 22.30% of practical queries are not well-designed patterns but still weakly monotonic.
- We consider three semantic features: monotonicity, non-monotonicity and weak monotonicity. And then we find that among these practical queries, monotonic queries is 65.61% and non-monotonic queries is about 0.04%. In other words, practical queries are almost weakly monotonic (about 99.96%). As a result, the evaluation of over 65.61% queries is in PTIME and the evaluation of over 99.96% queries is coNP-Complete while the evaluation with PSPACE-completeness involves in less 0.04% practical queries.
- We discuss an important semantic feature, namely satisfiability. We develop a sound algorithm to determine whether a given query with well-designed patterns is satisfiable or not. Finally, we can determine the satisfiability of all practical queries by analyzing common statistical structures of queries with non-monotonic well-designed patterns. As we expected, all practical queries are almost satisfiable (over 99.99%), which means the meaningless queries written by users are few in the real world.

The rest of this paper is organized as follows: Section 2 briefly introduces SPARQL 1.0. Section 3 discusses three statistic features. Section 4 and Section 5 discuss four semantic features. Finally, Section 6 summarizes the paper. Due to the limited space, we omit all proofs but a full technique report with consisting of all proofs can be found at a public website ¹.

2. SYNTAX AND SEMANTICS OF SPARQL

In this section, we briefly recall the syntax and semantics of SPARQL 1.0, largely depending on the excellent expositions [14, 19].

RDF graphs Let I , B and L be infinite sets of *IRIs*, *blank nodes* and *literals*, respectively. These three sets are pairwise disjoint. We denote the union $I \cup B \cup L$ by U and elements of $I \cup L$ will be referred to as *constants*. A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an *RDF triple*. An *RDF graph* is a finite set of RDF triples.

Patterns Assume furthermore an infinite set V of *variables*, disjoint from U . It is a SPARQL convention to prefix each variable with a question mark. Any triple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a pattern (called a *triple pattern*). *Patterns* are constructed by using triple patterns and operators UNION, AND, OPT, GRAPH and FILTER. Formally, patterns are of the forms as follows: P_1 UNION P_2 , P_1 AND P_2 , P_1 OPT P_2 , GRAPH _{i} (P), GRAPH _{γ_x} (P) and PFILTER C where $i \in I$ and C is a *constraint*. A constraint is a boolean combination of the 17 atomic constraints.

Semantics The semantics of patterns are defined in terms of sets of so-called *mappings*, which are simply total functions $\mu: S \rightarrow U$ on some finite set S of variables. We denote the domain S of μ by $\text{dom}(\mu)$. Now given an RDF graph G and a set of named graphs δ , $D = (G, \delta)$ denotes a RDF dataset. Given a pattern P , we define the semantics of P on D , denoted by $\llbracket P \rrbracket_D$, is a set of mappings whose satisfaction on constraints is based on a three-valued logic with truth values *true*, *false* and *error*.

Queries A SELECT query is an expression of the form SELECT _{S} P where S is a finite set of variables and P is a pattern. Semantically, given an RDF dataset D , we define $\llbracket \text{SELECT}_S P \rrbracket_D = \{\mu|_{\text{dom}(\mu) \cap S} \mid \mu \in \llbracket P \rrbracket_D\}$, where we use the common notation $f|_X$ for the restriction of a function f to a subset X of its domain.

3. STATISTIC FEATURES OF QUERIES

Our query log is extracted from LSQ [17] which is a public dataset of SPARQL queries extracted from the logs of public SPARQL endpoints. It contains 5.7 million query executions. LSQ is extracted from the following four SPARQL query logs: DBpedia (from 30/04/2010 to 20/07/2010, 232 million triples), Linked Geo Data (LGD) (from 24/11/2010 to 06/07/2011: 1 billion triples), Semantic Web Dog Food (SWDF) (from 16/05/2014 to 12/11/2014: 300 thousand triples) and British Museum (BM) (from 08/11/2014 to 01/12/2014: 1.4 million triples).

We firstly collect 1,749,069 unduplicated queries from 5,675,204 query executions. Secondly, we remove 555,084 queries which have parse error or do not follow the syntax of SPARQL 1.0. Indeed, it is still acceptable since only 0.6% queries are beyond SPARQL 1.0. Finally, we collect 1,087,544 (91.5% of SPARQL 1.0) SELECT queries by deleting all non-SELECT queries (i.e., ASK queries, DESCRIBE queries, or CONSTRUCT queries). So, in this paper, we mainly work on **1,087,544** queries from LSQ.

3.1 Occurrence frequency of triple patterns

The occurrence frequency of triple patterns is simply the number of triple patterns occurring in a query [15]. As a basic feature, it directly influences the size of queries. In our investigation, we see the followings:

- Over 54.99% queries contain only one triple pattern in one query;
- Most of practical queries contain at most 7 triple patterns (96%);
- The maximal number of triple patterns occurring in a query is 24.

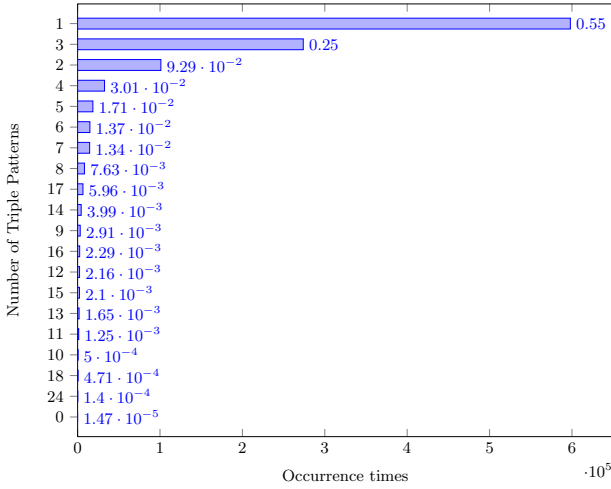
The full distribution of the occurrence frequency of triple patterns can be found in Figure 1.

This statistical result shows that a practical query generally contains a few triple patterns. It is helpful to restrict the scale of queries to be discussed. Since most of queries have

¹<http://123.56.79.184/Han2016TR.pdf>

few triple patterns, the shapes of graphs which are formed by the triple patterns are enumerable. So we can propose an optimized scheme for the enumerable shapes in the future.

Figure 1: Frequency of occurrences



3.2 Fragments

Though the complexity of a query depends on its occurrence frequency of triple patterns, the evaluation also relies on its grammatical structure. Therefore, it is necessary to take the grammatical structure of queries into account. Now, we investigate which fragments of SPARQL are popular among practical queries [17].

Let us abbreviate the operator AND by ‘A’; OPT by ‘O’; UNION by ‘U’; FILTER by ‘F’; and GRAPH by ‘G’. Then we can denote any fragment of SPARQL, where the letter word is formed by a subset of the five operators. We use ‘none’ to denote the fragment whose queries contain no operator.

We can find some interesting phenomena as follows:

- Over 37% queries are in the fragment of *none* which is the biggest fragment among all 32 fragments of SPARQL.
- Only 6 fragments: FGO, FGU, GOU, FGOU, AFGU and AFGOU, do not occur in our practical world. That is most of fragments are still useful.
- Total of four fragments: *none*, AO, F and A, are over 85.83% and there are over 94.8% if considering four fragments: AOU, FO, O, AFOU additionally.

As is well known, fragments of SPARQL have variational complexity of query evaluation. For instance, the complexity of query evaluation of AF is NP-complete even UNION is added [14]. In particular, the complexity of query evaluation becomes PSPACE-hard once OPT is added [14].

Thus, our statistical result could characterize the query evaluation of practical SPARQL queries precisely. For instance, the three fragments: *none*, A and F are over 62.83% whose query evaluation is PTIME. Moreover, it is interesting to optimize query evaluation of these fragments such as AO, F and A instead of the full SPARQL since they are popular in our practical world.

3.3 Well-designed patterns

Since OPT operator brings more complexity to query evaluation (generally, PSPACE-complete) [18] and these fragments with OPT operator are over 31.97% of the total. It is

interesting to discuss some restricted form of patterns with OPT in a lower complexity. The *well-designed* patterns [14] have been identified as a well-behaved class of SPARQL patterns, with properties similar to the conjunctive queries for relational databases. Thus the query evaluation of well-designed patterns becomes coNP-complete [2]. Let P be a pattern and C be a constraint, we use $var(P)$ to denote the set of variables occurring in P and $var(C)$ to denote the set of variables occurring in C .

A pattern Q is *safe* if for every subpattern P FILTER C of Q , it holds that $var(C) \subseteq var(P)$. A UNION-free pattern P is *well-designed* if P is safe and, for every subpattern $P' = (P_1 \text{ OPT } P_2)$ of P and for every variable $?x$ occurring in P_2 , the following condition holds: If $?x$ occurs both inside P_2 and outside P' , then it also occurs in P_1 .

A pattern is UNION *Normal Form* (UNF, for short) if it is in the form of $P_1 \text{ UNION } P_2 \text{ UNION } \dots \text{ UNION } P_n$, where each P_i ($1 \leq i \leq n$) is UNION-free.

A pattern in UNF is well designed if every P_i ($1 \leq i \leq n$) is a UNION-free well-designed pattern. A pattern is *non-well-designed* if it is not well-designed. Analogously, a query is *(non-)well-designed* if its pattern is *(non-)well-designed*. A query $\text{SELECT}_S(P)$ is called *well-designed* if P is well-designed.

Procedure of determining well-designed queries Now, we will present a procedure to determine whether a query is well-designed in three steps:

- Step 1** If a pattern contains the GRAPH operator or FILTER conditions which are not build-in condition, then it is not well-designed. In this case, there are 210,064/19.32% queries.
- If a pattern is not in the form of UNF then it is not well-designed. In this case, there are 32,598/3.00% queries.
 - If a pattern is a UNF pattern, then turn to **Step 2**. In this case, there are 6,745/0.62% queries.
 - If a pattern is a UNION-free pattern, then turn to **Step 3**. In this case, there are 838,137/0.62% queries.
- Step 2** For every UNION-free subpattern of a UNF pattern, using **Step 3**. If all the UNION-free subpatterns are well-designed, then the UNF pattern is well-designed (where there are 6,745/0.62% queries); otherwise, it is not well-designed.
- Step 3** If a pattern is a UNION-free well-designed pattern, then it is well-designed (where there are 837,839/77.04% queries); otherwise, it is not well-designed (where there are 298/0.03% queries).

We can conclude the following results:

- 844,584/77.66% queries are well-designed and the remaining 242,960/22.34% queries are not well-designed;
- Over 3/4 practical queries are *weak monotonicity* ([5], defined in Section 4) which provides a good support to well-designed patterns;
- According to above result, we can find that over 62.83% practical queries have query evaluation in PTIME while 14.83% practical queries have query evaluation in NP-complete and 22.34% practical queries have query evaluation in (possible) PSPACE-hard.

4. MONOTONICITY, WEAK MONOTONICITY AND NON-MONOTONICITY

In the statistics of well-designed patterns (see Section 3), we can find further some interesting phenomena as follows:

- over 62.83% practical queries with query evaluation in PTIME are monotonic;
- about 14.83% practical queries with query evaluation in NP-complete are not monotonic but weakly monotonic ([5], defined later);
- about 22.34% practical queries with query evaluation in (possible) PSPACE-Complete are possibly neither monotonic nor weakly monotonic.

In other words, three semantic features (i.e., monotonicity, weak monotonicity, non-monotonicity) connect with the complexity of query evaluation.

In this section, we look into the full distribution of monotonicity, weak monotonicity and non-monotonicity among our practical queries.

Monotonicity and weak monotonicity For every two RDF graphs G_1, G_2 such that $G_1 \subseteq G_2$, a pattern P is said to be *monotonic* if it holds that $\llbracket P \rrbracket_{G_1} \subseteq \llbracket P \rrbracket_{G_2}$. A pattern P is said to be *non-monotonic* otherwise. Let μ_1 and μ_2 be two mappings. μ_1 is *subsumed* in μ_2 denoted by $\mu_1 \sqsubseteq \mu_2$ if $\text{dom}(\mu_1) \subseteq \text{dom}(\mu_2)$ and $\mu_1(?x) = \mu_2(?x)$ for all $?x \in \text{dom}(\mu_1)$. Let Ω_1 and Ω_2 be two sets of mappings. For any mapping $\mu_1 \in \Omega_1$, we use $\Omega_1 \sqsubseteq \Omega_2$ if there exists some mapping $\mu_2 \in \Omega_2$ such that $\mu_1 \sqsubseteq \mu_2$. For any two RDF graphs G_1 and G_2 , a pattern P is *weakly monotonic* if $G_1 \subseteq G_2$ implies $\llbracket P \rrbracket_{G_1} \sqsubseteq \llbracket P \rrbracket_{G_2}$. Note that all monotonic queries are weakly monotonic but not vice versa and all weakly monotonic queries are non-monotonic but not vice versa.

In the following of this paper, we mainly count weakly monotonic queries by excluding monotonic queries, as is well known, monotonic queries are weakly monotonic.

Determining monotonicity, weak monotonicity and non-monotonicity Since $\text{GRAPH}_v(P)$ and $P \text{ FILTER } C$ have the same monotonicity (weak monotonicity or non-monotonicity) as P [2], we can ignore the GRAPH operator and the difference between the non-build-in condition and build-in condition. Moreover, queries in OPT-free fragments are monotonic. Thus we can exclude these queries.

Main procedure The main procedure contains the following steps:

Step 1 For every pattern, if it is not in UNF, we need to rewrite it to its UNF according to [2, Lemma 3] (where there are 44,101/4.06%).

Step 2 For every pattern in UNF, if all the UNION-free sub-patterns in it are OPT-free or OPT-monotonic then it is monotonic. In this case, there are 713,532/65.61% queries. Otherwise, if all the UNION-free sub-patterns are UNION-free well-designed sub-patterns then the pattern is weakly monotonic [2]. In this case, there are 330,389/30.38% queries.

A UNION-free pattern P is *OPT-monotonic* if $\text{vars}(P_2) \subseteq \text{vars}(P_1)$ for every subpattern $Q = P_1 \text{ OPT } P_2$ of P .

Immediately, we can conclude the following proposition.

PROPOSITION 4.1. *Let P be a UNION-free pattern. If P is OPT-monotonic then P is monotonic.*

Exceptions of determining procedure In the first step of main procedure, not all patterns can be logically trans-

lated into their UNION normal form due to the distributive law on $P_1 \text{ OPT } (P_2 \text{ UNION } P_3)$ disabled (about 20,361/1.87% queries), where there are four cases: (1) $(?x, p, ?y) \text{ OPT } ((?x, q, ?z) \text{ UNION } (?x, r, ?u) \text{ UNION } (?x, s, ?v))$ (18,466/1.70%) (weakly monotonic); (2) $(?x, p, a) \text{ OPT } ((b, q, ?y) \text{ UNION } (c, r, ?y))$ (1,494/0.14%) (weakly monotonic); (3) $(?x, p, a) \text{ OPT } ((?x, q, ?y) \text{ UNION } (?z, r, b))$ (388/0.04%) (weakly monotonic); and (4) $((?x, p, q) \text{ OPT } (?y, q, a)) \text{ OPT } ((?y, q, b) \text{ UNION } (?z, r, c))$ (13/0.00%) (non-monotonic).

Finally, among these queries which can be rewritten into their UNION normal forms, there still exist 23,262/2.14% SPARQL queries that are neither well-designed nor can be rewritten to well-designed queries.

The remaining unknown SPARQL queries contain the following five cases: (1) $((?x, p, a) \text{ OPT } (?x, q, ?y)) \text{ FILTER } (\text{langMatches}(\text{lang}(?y), 'en'))$ (22,848/2.10%) (montone); (2) $((\text{OPT}(?x, p, a)) \text{ OPT } (?x, q, b))$ (77/0.01%) (non-monotonic); (3) $((?x, a, b) \text{ OPT } (?x, c, ?y)) \text{ FILTER } \neg \text{bound}(?y)$ (213/0.02%) (non-monotonic); (4) $((?x, a, b) \text{ OPT } (?y, c, d)) \text{ OPT } (?y, d, e)$ (120/0.01%) (non-monotonic); and (5) $((?x, p, ?y) \text{ OPT } (?y, q, ?z)) \text{ FILTER } (\neg \text{bound}(?z)) \vee (?x = a)$ (1/0.00%) (non-monotonic).

At last, there are three SPARQL queries which are not safe. Since the queries are not safe, they are not satisfiable [24] (discussed in Section 5). So, by default, these queries are monotonic.

Statistical results We can show the result in the following:

- It is a surprise that over 713,548/65.61% queries are monotonic and about 373,578/34.35% queries are weakly monotonic while only 418/0.04% queries are non-monotonic. In other words, over 99.96% queries are weakly monotonic.
- In the weakly monotonic fragment(1,087,126 queries), there are 844,584/77.69% queries are well-designed patterns. Moreover, there are about 242,542/22.31% weakly monotonic queries which are not well-designed patterns.
- In the non-well-designed fragment(242,996 queries), over 222,194/91.45% queries which are not well-designed can be logically translated into equivalent well-designed patterns and about 20,348/8.38% queries which are neither well-designed patterns nor can be rewritten to well-designed patterns but are still weakly monotonic. Otherwise, there are 418/0.17% queries which are non-monotonic.

5. SATISFIABILITY OF QUERIES

Although the satisfiable problem of well-designed patterns is decidable, this problem of the full SPARQL is undecidable [24]. Base on this, we are interested to know how many queries are satisfiable. Note that a query is called *satisfiable* if there exists an RDF graph under which the pattern evaluates to a nonempty set of mappings.

Procedure of determining satisfiability The main procedure of determining whether a query is satisfiable consists of six steps as follows:

Step 1 If $\llbracket P \rrbracket_G \neq \emptyset$ for some RDF graphs in existing dataset then P must be satisfiable, otherwise turn to **Step 2**. In this pre-processing step, we can process 635,704/58.46% queries.

Step 2 If P is a filter-free pattern then it is satisfiable [24], otherwise turn to **Step 3**. In this step, we can handle

further 372753/34.28% queries.

Step 3 If P contains negated bound constraint then put it into a pool to be determined at last (about 84 queries); otherwise enter **Step 4**.

Step 4 If P is well-designed or can be rewritten to a well-designed pattern then turn to **Step 5**; otherwise again put it into a pool to be determined last (about 13,630/1.25% queries).

Step 5 Using Algorithm 1 to determine its satisfiability of the remaining queries (about 65,373/6.01%).

Since the satisfiability of a well-designed pattern is equivalent to an OPT-free pattern [24, Proposition 1], we mainly consider the satisfiability of OPT-free patterns in the following of this section. To determine the satisfiability of well-designed AF-queries, we need the following three sub-steps of **Step 6**:

Step 6.1 Rewrite constraints exhaustively by applying all inference rules in Table 1.

Step 6.2 Translate all patterns into its *strong UNION normal form* (strong UNF, for short) where a pattern is of the form $Q_1 \text{ UNION } \dots \text{ UNION } Q_m$, where each Q_i ($1 \leq i \leq m$) is an AF-pattern and all constraints occurring in Q_i are atomic.

Step 6.3 Determining the closeness of the closure (i.e., collection) which includes all atomic constraints. If it is close then return “unsatisfiable”; otherwise return “satisfiable”. A set of constraints is *close* if it subsumes a *conflict* of the form $\{\alpha, \neg\alpha\}$ for an atomic constraint α .

PROPOSITION 5.1. *Let C be a constraint. For any pattern P , $P \text{ FILTER } C \equiv P \text{ FILTER } C'$ where C' is obtained from C by using inference rules in Table 1. Note that, all the atomic constraints can infer itself, we omit the inference from it to itself.*

By Proposition 5.1, we can conclude that all OPT-free patterns can be logically translated into its strong UNION normal form.

PROPOSITION 5.2. *Let P be an OPT-free pattern. There exists some Q in strong UNF such that $P \equiv Q$.*

Now, we conclude that Algorithm 1 is sound and complete.

THEOREM 5.1. *Let P be an AF-pattern. If P is in strong UNION normal form then P is satisfiable iff the closure of P obtained in Algorithm 1 is not close.*

At last, the remaining queries (about 13,630/1.25%), which are neither well-designed patterns nor able to be logically translated to equivalent well-designed queries, contains the following cases: (1) $((?x, a, b) \text{OPT}(?y, c, d)) \text{FILTER } \neg \text{bound}(?y)$ (84/0.01%) (unsatisfiable); (2) $(P_1 \text{OPT } P_2 \text{OPT } \dots \text{OPT } P_n)$ (13,488/(1.24%)) (as the same as P_1); and (3) $(?x, a, b) \text{FILTER bound}(?y)$ (58/0.00%) (satisfiable).

As we expected, over 99.99% of practical queries are satisfiable, that is, the meaningless queries written by users are few in our practical world.

Zhang and Van den bussche[24] presented a determination method of satisfiability for well-designed patterns. In this paper, we implement an algorithm to determine the satisfiability of practical queries.

Algorithm 1 Determining the satisfiability of well-designed pattern

Input: Well-designed pattern P in strong UNION normal form

Output: Determining the satisfiability of P

```

1:  $P = (Q_1 \text{ UNION } Q_2 \text{ UNION } \dots \text{ UNION } Q_m)$ 
2: for every  $Q_j (1 \leq j \leq m)$  in  $P$  do
3:    $S = \{C_1, C_2, \dots, C_k\}$ , ( $1 \leq j \leq k$ ,  $C_j$  is constraint in  $Q_j$ )
4:    $\mathcal{L} = \{S\}$ 
5:   repeat
6:     for every  $S_i$  in  $\mathcal{L}$  do
7:       if  $\text{isChanged}_i = \text{false}$  then
8:         continue;
9:       else
10:         $\text{isChanged}_i = \text{false}$ 
11:      end if
12:      for every  $C_j$  in  $S_i$  do
13:        if  $C_j \rightarrow D$  and  $D \notin S_i$  then
14:           $S_i = S_i \cup D$ ,  $\text{isChanged}_i = \text{true}$ 
15:        end if
16:      end for
17:      if  $C_j = C_1 \wedge C_2$  then
18:        if  $\{C_1, C_2\} \not\subseteq S_i$  then
19:           $S_i = S_i \cup \{C_1, C_2\}$ ,  $\text{isChanged}_i = \text{true}$ 
20:        end if
21:      end if
22:      if  $C_j = C_1 \vee C_2$  then
23:        if  $\{C_1, C_2\} \cap S_i = \emptyset$  then
24:           $S_i = S_i \cup \{C_1\}$ ,  $S'_i = S_i \cup \{C_2\}$ ,  $\mathcal{L} =$ 
            $\mathcal{L} \cup \{S'_i\}$ ,  $\text{isChanged}_i = \text{true}$ 
25:        end if
26:      end if
27:    end for
28:  until every  $\text{isChanged}_i$  is false
29:   $\mathcal{L} = \{S_1, S_2, \dots, S_l\}$ 
30:  if there exists  $S_i (1 \leq i \leq l)$  is consistent then
31:    return  $P$  is satisfiable.
32:  end if
33: end for
34: return  $P$  is unsatisfiable.

```

6. CONCLUSIONS

In this paper, we have presented comprehensive statistics of seven basic features of a log of SPARQL 1.0 queries in LSQ. We think that these statistical results could provide some useful observation for researchers and engineers who are interested in what practical SPARQL queries look like. In the future, we are going to analyze other interesting features of SPARQL 1.0 queries.

7. REFERENCES

- [1] Arias M., D. Fernández J., A. Martínez-Prieto M., de la Fuente P. (2011). An empirical study of real-world SPARQL queries. In: *Proc. of USEWOD'11 in WWW'11*.
- [2] Arenas M., Pérez J. (2011). Querying semantic web data with SPARQL, In: *Proc. of PODS'11*, pp. 305–316.
- [3] Barceló P., Pichler R., Skritek S. (2015) Efficient evaluation and approximation of well-designed pattern

trees, In: *Proc. of PODS'15*, pp.131–144.

- [4] Cebiric S., Goasdoué F., Manolescu I.(2015). Query-oriented summarization of RDF graphs. In: *Proc. of VLDB'15*, PVLDB,8(12): 2012–2023.
- [5] Chandra A.K., Harel D. (1980). Computable queries for relational data bases, *J. Comput. Syst. Sci.*, 21(2):156–178.
- [6] Guido N. (2015). On the static analysis for SPARQL queries using modal logic. In: *Proc. of IJCAI'15*, pp. 4367–4368.
- [7] Harbi R., Abdelaziz I., Kalnis P., Mamoulis N. (2015). Evaluating SPARQL queries on massive RDF datasets. In: *Proc. of VLDB'15*, PVLDB, 8(12): 1848–1859.
- [8] Hellings J., Kuijpers B., Van den Bussche J., Zhang X.(2013). Walk logic as a framework for path query languages on graph databases, In: *Proc. of ICDT'13*, pp.117–128.
- [9] Letelier A., Pérez J., Pichler R., Skritek S.(2013). Static analysis and optimization of semantic web queries, *ACM Trans. Database Syst.*, 38(4): article 25.
- [10] Letelier A., Pérez J., Pichler R., Skritek S.(2012). SPAM: A SPARQL analysis and manipulation tool, In: *Proc. of VLDB'12*, PVLDB, 5(12):1958–1961.
- [11] Morsey M., Lehmann J., Auer S., Ngonga Ngomo A.(2011) DBpedia SPARQL benchmark - performance assessment with real queries on real data, In: *Proc. of ISWC'11*, pp. 454–469.
- [12] Neumann T., Weikum G.(2009). Scalable join processing on very large RDF graphs. In: *Proc. of SIGMOD'09*, pp.627–IC640.
- [13] Neumann T., Weikum G.(2010). The RDF-3X engine for scalable management of RDF data. In: *Proc. of VLDB'10*, PVLDB, 19(1):91C113.
- [14] Pérez J., Arenas M., Gutierrez C.(2009). Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):article 16.
- [15] Picalausa F., Vansummeren S.(2011). What are real SPARQL queries like? In: *Proc. of SWIM'11*, article 7.
- [16] RDF primer, W3C Recommendation, February 2004.
- [17] Saleem M., Intizar Ali M., Hogan A., Mehmood Q., Ngonga Ngomo, A. (2015). LSQ: The linked SPARQL queries dataset, In: *Proc. of ISWC'15*, pp.261–269.
- [18] Schmidt M., Meier M., Lausen G.(2010). Foundations of SPARQL query optimization, In: *Proc. of ICDT'10*, pp.4–33.
- [19] SPARQL query language for RDF, W3C Recommendation, January 2008.
- [20] SPARQL 1.1 query language, W3C Recommendation, March 2013.
- [21] Wood P. (2012). Query languages for graph databases, *SIGMOD Record*, 41(1): 50–60.
- [22] Zhang X., Van den Bussche J.(2014). On the primitivity of operators in SPARQL. *Inf. Process. Lett.*, 114(9): 480-485
- [23] Zhang X., Van den Bussche J.(2015). On the power of SPARQL in expressing navigational queries, *Computer J.*, 58 (11): 2841-2851.
- [24] Zhang X., Van den Bussche J.(2014). On the satisfiability problem for SPARQL patterns, arXiv:1406.1404.

Table 1: Inference Rules

Atomic constraints	Inferred constraints
$?x = c$	$\text{bound}(?x)$
$?x \neq c$	$\text{bound}(?x)$
$?x = ?y$	$\text{bound}(?x) \wedge \text{bound}(?y)$
$?x \neq ?y$	$\text{bound}(?x) \wedge \text{bound}(?y)$
$?x > c$	$((?x \geq c) \wedge (?x \neq c)) \wedge \text{isLiteral}(?x) \wedge (\neg \text{isBlank}(?x) \wedge \neg \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$?x \leq c$	$((?x < c) \vee (?x = c)) \wedge \text{isLiteral}(?x) \wedge (\neg \text{isBlank}(?x) \wedge \neg \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$?x < c$	$((?x \leq c) \wedge (?x \neq c)) \wedge \text{isLiteral}(?x) \wedge (\neg \text{isBlank}(?x) \wedge \neg \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$?x \geq c$	$\wedge((?x > c) \vee (?x = c)) \wedge \text{isLiteral}(?x) \wedge (\neg \text{isBlank}(?x) \wedge \neg \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$\text{isLiteral}(?x)$	$(\neg \text{isBlank}(?x) \wedge \neg \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$\neg \text{isLiteral}(?x)$	$(\text{isBlank}(?x) \vee \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$\text{isBlank}(?x)$	$(\neg \text{isLiteral}(?x) \wedge \neg \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$\neg \text{isBlank}(?x)$	$(\text{isLiteral}(?x) \vee \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$\text{isIRI}(?x)$	$(\neg \text{isBlank}(?x) \wedge \neg \text{isLiteral}(?x)) \wedge \text{bound}(?x)$
$\neg \text{isIRI}(?x)$	$(\text{isLiteral}(?x) \vee \text{isBlank}(?x)) \wedge \text{bound}(?x)$
$\text{str}(?x) = c$	$(\text{isLiteral}(?x) \vee \text{isIRI}(?x)) \wedge \neg \text{isBlank}(?x) \wedge \text{bound}(?x)$
$\text{str}(?x) \neq c$	$(\text{isLiteral}(?x) \vee \text{isIRI}(?x)) \wedge \neg \text{isBlank}(?x) \wedge \text{bound}(?x)$
$\text{lang}(?x) = c$	$\text{langMatches}(\text{lang}(?x), c) \wedge \text{isLiteral}(?x) \wedge (\neg \text{isBlank}(?x) \wedge \neg \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$\text{lang}(?x) \neq c$	$\neg \text{langMatches}(\text{lang}(?x), c) \wedge \text{isLiteral}(?x) \wedge (\neg \text{isBlank}(?x) \wedge \neg \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$\text{langMatches}(\text{lang}(?x), c)$	$\text{lang}(?x) = c \wedge \text{isLiteral}(?x) \wedge (\neg \text{isBlank}(?x) \wedge \neg \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$\neg \text{langMatches}(\text{lang}(?x), c)$	$(\text{lang}(?x) \neq c) \wedge \text{isLiteral}(?x) \wedge (\neg \text{isBlank}(?x) \wedge \neg \text{isIRI}(?x)) \wedge \text{bound}(?x)$
Regex($\text{str}(?x), r$)	$(\text{isLiteral}(?x) \vee \text{isIRI}(?x)) \wedge \neg \text{isBlank}(?x) \wedge \text{bound}(?x)$
$\neg \text{Regex}(\text{str}(?x), r)$	$(\text{isLiteral}(?x) \vee \text{isIRI}(?x)) \wedge \text{bound}(?x) \wedge \neg \text{isBlank}(?x)$
Regex($?x, r$)	$\text{Regex}(\text{str}(?x), r) \wedge \text{isLiteral}(?x) \wedge (\neg \text{isBlank}(?x) \wedge \text{isIRI}(?x)) \wedge \text{bound}(?x)$
$\neg \text{Regex}(?x, r)$	$(\neg \text{Regex}(\text{str}(?x), r) \wedge \text{isLiteral}(?x) \wedge (\neg \text{isBlank}(?x) \wedge \neg \text{isIRI}(?x)) \wedge \text{bound}(?x)$